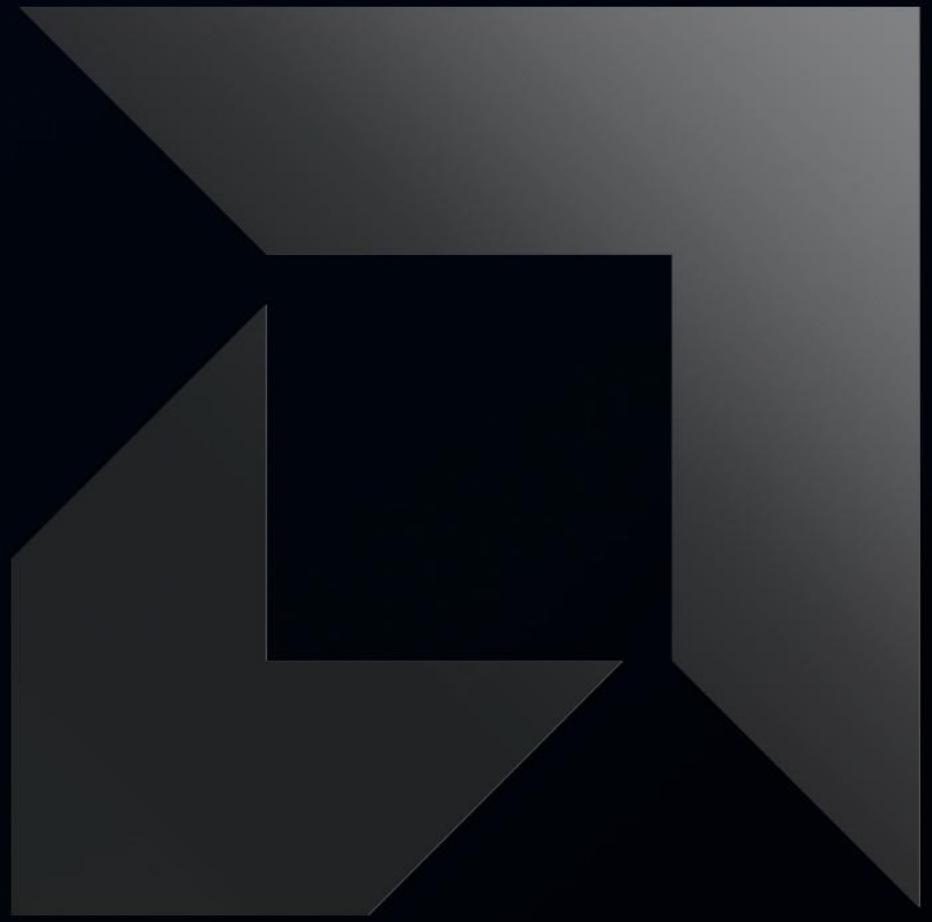


Introduction to OpenMP® Offload on AMD GPUs

Presenter: Bob Robey
AMD @ Tsukuba University
Oct 21-23, 2025

AMD
together we advance_

Motivation for OpenMP for GPUs

Fortran:

```
!$omp parallel do private(i) shared(Nl,…)
do i=1,Nl
  …
end do
!$omp end parallel
```

CPU examples

C/C++:

```
#pragma omp parallel for private(i) shared(Nl,…)
for(int i=0, i<Nl, i++){
…
}
```

- Why use OpenMP for porting to GPUs?
  - ✓ OpenMP is standardized
  - ✓ Portable code
  - ✓ Fortran, C, and C++ supported
- Many HPC codes are already OpenMP (+MPI) parallelized on CPUs
  - ✓ Porting requires only few code changes
  - ✓ Easy to learn
- Interoperability with HIP and ROCm libraries
  - ✓ Flexibility
- GPU code can be compiled for the CPU, too
  - ✓ Start porting before having access to GPUs
  - ✓ Majority of correctness checks possible without access to GPUs

Compilers – two major strands

Primary Support

- LLVM™ based
  - ROCm compilers provide support for both HIP and OpenMP®
    - hipcc -- /opt/rocm/bin
    - amdclang -- /opt/rocm/llvm/bin
  - AOMP: AMD OpenMP® research compiler for prototyping new features

Use LLVM based compilers for production (or Cray if available)

Functionality Only

- GCC based -- GNU Compilers
  - Provide offloading support to AMD GPUs (OpenMP®, OpenACC)
    - GCC 11 added offloading for the AMD MI100 GPUs
    - GCC 13 adds support for the AMD MI200 GPU series.
  - https://gcc.gnu.org/wiki/Offloading

amdflang-new beta pre-release of AMD Fortran compiler

Installed on aac6 to use in this training:

module load amdflang-new/

Module will set the following:

export FC=amdflang-new

AMD is working on a Fortran compiler and is ready to share a beta version. During the training, you will have hands-on access to the beta. Going forward, AMD will continue to provide early access via AFARs (Advanced Feature Access Releases) as improvements and features are added to the beta. **As with any beta, we are looking to gather feedback on functionality, usability and user experience.**

New Flang is released in the ROCm 7.0 version!

Blog post about next gen Fortran compiler: https://rocm.blogs.amd.com/ecosystems-and-partners/fortran-journey/README.html

PRE-PRODUCTION SOFTWARE: The software accessible on this training may be a **pre-production version**, intended to provide advance access to features that may or may not eventually be included into production version of the software. Accordingly, pre-production software **may not be fully functional, may contain errors, and may have reduced or different security, privacy, accessibility, availability, and reliability standards** relative to production versions of the software. Use of pre-production software may result in unexpected results, loss of data, project delays or other unpredictable damage or loss. Pre-production software is not intended for use in production, and your use of pre-production software is at your own risk.

Enabling OpenMP® on AMD Hardware

		LLVM		GCC	
Compiler Module ➡		amdclang/aomp/clacc		gcc/og	
		Command	Flags	Command	Flags
Language	C	amdclang clang	-fopenmp --offload-arch=<gfx###>	gcc	-fopenmp --foffload=-march=<gfx###>
	C++	amdclang++ clang++	-fopenmp --offload-arch=<gfx###>	g++	-fopenmp --foffload=-march=<gfx###>
	Fortran	amdflang(-new) flang(-new)	-fopenmp --offload-arch=<gfx###>	gfortran	-fopenmp --foffload=-march=<gfx###>

Offloading Target (CPU/GPU/GCD)	Architecture <gfx###>
AMD MI300 Series	gfx942
AMD MI200 Series	gfx90a
AMD MI100	gfx908
Native Host (CPU)	-fopenmp-targets=amdgcn-amd-amdhsa

Sample OpenMP® Makefile

```
EXEC = reduction
default: ${EXEC}
all: ${EXEC}

ROCM_GPU ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))

CC1=$(notdir $(CC))

ifeq ($(findstring amdclang,$(CC1)), amdclang)
  OPENMP_FLAGS = -fopenmp --offload-arch=$(ROCM_GPU)
else ifeq ($(findstring clang,$(CC1)), clang)
  OPENMP_FLAGS = -fopenmp --offload-arch=$(ROCM_GPU)
else ifeq ($(findstring gcc,$(CC1)), gcc)
  OPENMP_FLAGS = -fopenmp -foffload=-march=$(ROCM_GPU)
else ifeq ($(findstring cc,$(CC1)), cc)
  OPENMP_FLAGS = -fopenmp
  #the cray compiler decides the offload-arch by loading appropriate modules
  #module load craype-accel-amd-gfx942 for example
endif

CFLAGS = -g -O3 -fstrict-aliasing ${OPENMP_FLAGS}
LDFLAGS = ${OPENMP_FLAGS} -fno-lto -lm

${EXEC}: ${EXEC}.o codelet.o
        $(CC) $(LDFLAGS) $^ -o $@

# Cleanup
clean:
        rm -f *.o ${EXEC}
```

Sample OpenMP® CMakeLists

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Memory_pragmas LANGUAGES CXX)

set (CMAKE_CXX_STANDARD 17)

if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
endif(NOT CMAKE_BUILD_TYPE)

string(REPLACE -O2 -O3 CMAKE_CXX_FLAGS_RELWITHDEBINFO ${CMAKE_CXX_FLAGS_RELWITHDEBINFO})
set(CMAKE_CXX_FLAGS_DEBUG "-ggdb")
if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
   # using Clang
   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp --offload-arch=gfx942 -fstrict-aliasing")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU")
   # using GCC
   set(CMAKE_CXX_FLAGS
       "${CMAKE_CXX_FLAGS} -fopenmp -foffload=-march=gfx942 -fstrict-aliasing -fopt-info-optimized-omp")
elseif (CMAKE_CXX_COMPILER_ID MATCHES "Cray")
endif()

add_executable(mem1 mem1.cc)
```

How we begin

- The first step in porting an application to use OpenMP® for offloading work to a GPU is to add a compute pragma/directive to a loop
- Choose one of the more expensive loops in your application for the first loop to add a compute pragma
  - This will give the most compute speedup to offset the slowdown from memory transfers to the GPU and back
  - Testing OpenMP® parallelism on the CPU can identify potential problems
  - Another quick test is to reverse the loop direction on a loop to see if there is an order required for correctness
- A profile of your existing code will help identify potential loops to offload to the GPU
- Pay attention to what data needs to be moved to enable the offloading

- While this appears simple, many complex physics applications can be difficult to approach
- You may want to try simpler loops first just to get some experience with the process

Running example for this presentation: saxpy

Declaration and
Initialization of arrays on CPU
in main routine (not shown)

```
void saxpy(float a, float *x, float *y, int N) {

#pragma omp parallel for
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

   printf("check output:\n");
   printf("y[0] %lf\n",y[0]);
   printf("y[N-1] %lf\n",y[N-1]);
}
```

This is the code we want to execute on a target device (i.e., GPU) with appropriate directives to be introduced next

This is for CPU: OpenMP® Pragma to run in parallel with multiple threads

WARNING: operations like saxpy should be done through an efficient library (e.g. hipblas) in production codes

If you prefer Fortran

```
subroutine saxpy(a, x, y, n)
   use iso_fortran_env
   implicit none
   integer,intent(in) :: n
   real(kind=real32),intent(in) :: a
   real(kind=real32), dimension(:),allocatable,intent(in) :: x
   real(kind=real32), dimension(:),allocatable,intent(inout) :: y
   integer :: i

   !$omp parallel do simd
   do i=1,n
       y(i) = a * x(i) + y(i)
   end do

   write(*,*) "plausibility check:"
   write(*,'("y(1) ",f8.6)') y(1)
   write(*,'("y(n-1) ",f8.6)') y(n-1)
end subroutine saxpy
```

Declaration and
Initialization of arrays on CPU
in main routine (not shown)

This is the code we want to execute on a target device (i.e., GPU) with appropriate directives to be introduced next

This is for CPU: OpenMP® directive to run in parallel with multiple threads

How to offload to the GPU: the `target` directive

- Transfer **control and data** from the host to the device
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom}:] list)
if(scalar-expr)
```

NOTE: target will move the computation to the GPU, but **single thread only** instead of using the whole device. Occupancy will be <1%

But there is more to a GPU compute directive than just “target”

- We need to tell the compiler what hardware resources to use
- We also need to tell it how to spread out the work
- Sometimes the compiler can detect information and do the right thing, other times it cannot

- Despite all this possible complexity, most compute pragmas are simple one-liners to use all the hardware and distribute all the work

- `#pragma omp target teams distribute parallel for simd`

  or for Fortran

- `!$omp target teams distribute parallel do simd`

- C/C++ uses `pragmas` as directives to the compiler
- Fortran uses a comment syntax for its directives

- We‘ll go into each part of this pragma in the next talk

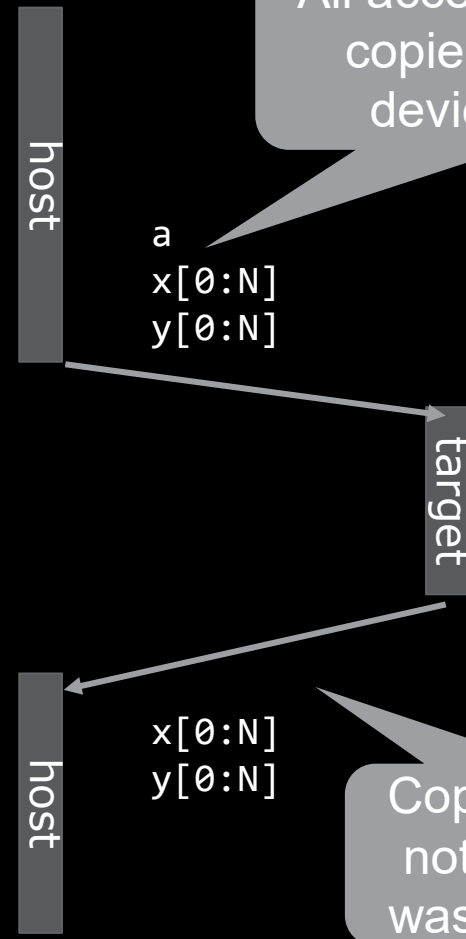
Example: saxpy_gpu_singleunit_static

Compiler does the work

```
int main(int argc, char *argv[]){
   int N=100000;
   float a=2.0f;
   float x[N], y[N];

   for (int i = 0; i < N; i++) {
      x[i] = 1.0f; y[i] = 2.0f;
   }

#pragma omp target teams distribute parallel for simd
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

   printf("check output:\n");
   printf("y[0] %lf\n",y[0]);
   printf("y[N-1] %lf\n",y[N-1]);
}
```

The compiler identifies variables that are used in the target region

All accessed arrays are copied from host to device and back.



Copying x back is not necessary. It was not changed.

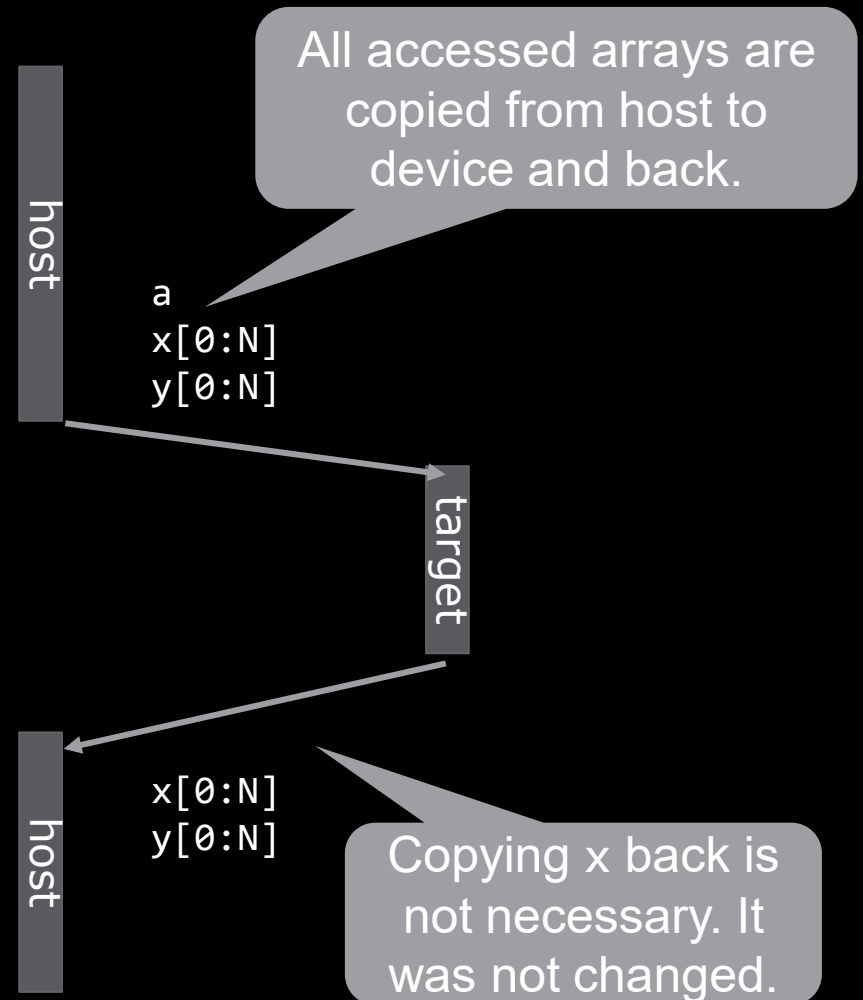
Example: saxpy_gpu_singleunit_dynamic

```
int main(int argc, char *argv[]){
   int N=10000000;
   float a=2.0f;

   float *x = (float *)malloc(N*sizeof(float));
   float *y = (float *)malloc(N*sizeof(float));

   for (int i = 0; i < N; i++) {
      x[i] = 1.0f; y[i] = 2.0f;
   }

#pragma omp target teams distribute parallel for simd
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

   printf("check output:\n");
   printf("y[0] %lf\n",y[0]);
   printf("y[N-1] %lf\n",y[N-1]);

   free(x); free(y);
}
```

The compiler identifies variables that are used in the target region but cannot get the sizes

Requires "export HSA_XNACK=1" or map clause

In Fortran allocatable arrays are mapped by default



A note on the simd clause

- C/C++
  - `#pragma omp target teams distribute parallel for simd`
- Fortran
  - `!$omp target teams distribute parallel do simd`

- Note on the compute construct:
  - The `simd` clause doesn't do anything with the AMD OpenMP compiler and can be dropped.
  - Nvidia also doesn't use it.
  - Cray used to be the most common compiler where it was used, but they are moving to drop it as well.
  - But…there are still OpenMP compilers such as the Intel® compiler that recognize it and use it.
  - So…for maximum portability, we include it. But you may want to drop it depending on the systems you plan to run on.
  - If you use the `simd` clause, you will get warnings about not being able to vectorize during compilation. These are harmless and can be ignored.

Other compute clauses -- loop

- The loop clause was added as a simpler option that gives the compiler more freedom in implementing the parallelism for a for loop

  ```
  #pragma omp target teams loop
  ```

  - Reduces the pragma complexity
  - Compilers may not optimize (or implement) this case as well because it is new.

- ROCm will generate an optimized target region implementation for AMD GPUs to use

  ```
  #pragma omp target teams loop
  ```

  Instead of

  ```
  #pragma omp target teams distribute parallel for
  ```

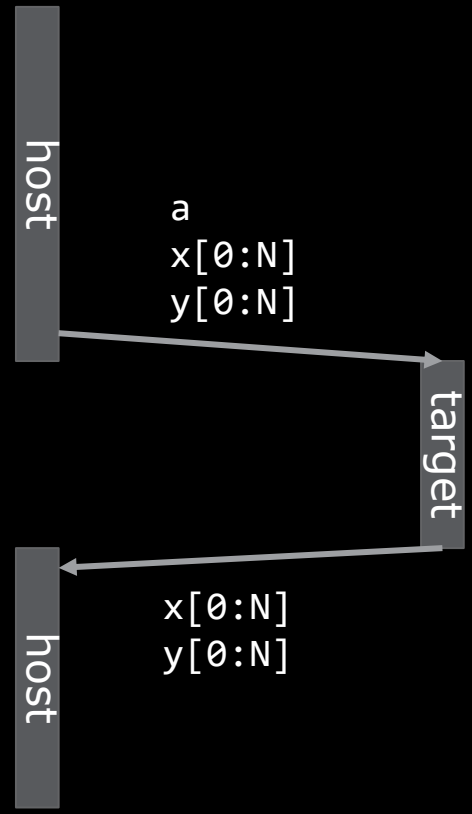
  In OpenMP 6.0, replaces `teams distribute parallel for`

- The loop clause will be coming soon for the AMD next generation flang compiler.

Example: saxpy

Simpler “loop” pragma/directive is equivalent to
`#pragma omp target teams distribute parallel do simd`

```
void saxpy(float a, float *x, float *y, int N) {
#pragma omp target teams loop
   for (int i = 0; i < N; i++) {
      y[i] += a * x[i];
   }

   printf("check output:\n");
   printf("y[0] %lf\n",y[0]);
   printf("y[N-1] %lf\n",y[N-1]);
}
```




```
amdclang -fopenmp --offload-arch=$GPU_ARCH ...
```

Nested Loops – collapse clause

- `collapse(n)` clause will collapse nested loops into one parallel construct. It can generate more work groups to help fill up the GPUs
  - Originally, only perfectly nested loops could be collapsed (no extra lines of code between the for/do loops)
  - This constraint has been relaxed and some lines of code are now acceptable both before and after the inner loop
  - This example can be collapsed by the ROCm LLVM™ OpenMP® compiler (and LLVM trunk)

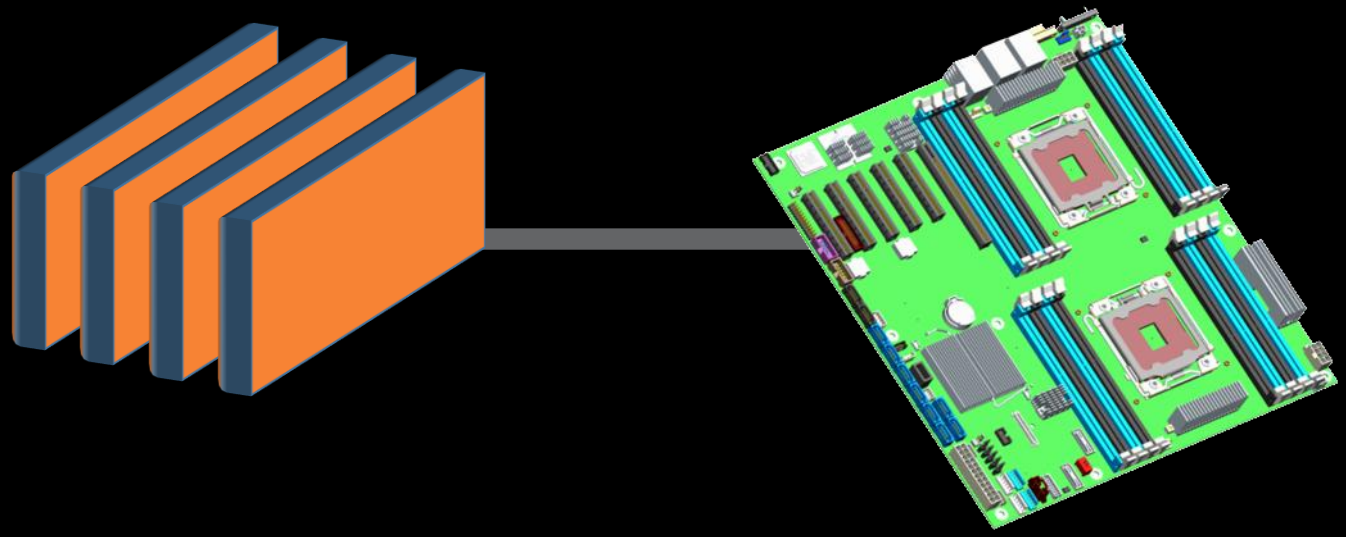
```
int A[N][N];
int B[N][N];
int e = -1;
#pragma omp target teams distribute
parallel for collapse(2)
for (int j = 0; j< N; j++) {
    e++;
    for (int i = 0; i < N; i++)
        a[j][i]=b[j][i]+e;
    e++;
}
```

Nested Loops – collapse clause example

```
subroutine saxpy(a, x, y, m, n)
   use iso_fortran_env
   implicit none
   integer,intent(in) :: m, n
   real(kind=real32),intent(in) :: a
   real(kind=real32), dimension(:,:),allocatable,intent(in) :: x
   real(kind=real32), dimension(:,:),allocatable,intent(inout) :: y
   integer :: i, j
   real(kind=real64) :: start, finish

   !$omp target teams distribute parallel do collapse(2)
   do j=1,n
     do i=1,m
       y(i,j) = a * x(i,j) + y(i,j)
     end do
   end do

   write(*,*) "plausibility check:"
   write(*,'("y(1,1) ",f8.6)') y(1,1)
   write(*,'("y(m,n) ",f8.6)') y(m,n)
end subroutine saxpy
```

Collapse clause causes the work to be distributed from both loops

Optimizing Data Transfers is Key to Performance on discrete GPUs

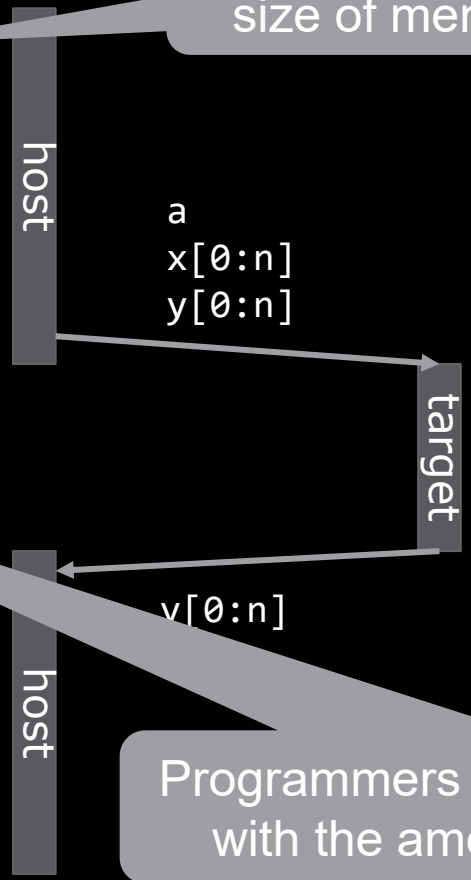


- Connections between host and accelerator are typically lower-bandwidth, higher-latency interconnects
  - Bandwidth host memory: hundreds of GB/sec
  - Bandwidth accelerator memory: TB/sec
  - PCIe® Gen 4 bandwidth (16x): tens of GB/sec

- Unnecessary data transfers must be avoided, by
  - only transferring what is actually needed for the computation, and
  - making the lifetime of the data on the target device as long as possible.

Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int n) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:n]) \
                       map(tofrom:y[0:n])
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



The compiler cannot determine the size of memory behind the pointer.

Programmers have to help the compiler with the amount of data to transfer.

OpenMP® Device Constructs: `target data`

- Create scoped data environment and transfer data from the host to the device and back
- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],…]
structured-block
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom | release | delete}:] list)
if(scalar-expr)
```

Optimize Data Transfers

- Reduce the amount of time spent transferring data
  - Use `map` clauses to enforce direction of data transfer.
  - Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device (see next slide).

Create data environment.

```
void example() {
    float tmp[N], a[N], b[N], c[N];
    #pragma omp target data map(alloc:tmp[:N])  \
                            map(to:a[:N],b[:N]) \
                            map(tofrom:c[:N])
    {
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N); // uses target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses target
        saxpy(2.0f, c, tmp, N);
}   }
```

No `map` clauses! Presence checks will find data via the pointer.

```
void zeros(float* a, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}

void saxpy(float a, float* y, float* x, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

Unstructured Data Regions

- Some programming styles such as C++ classes are difficult to enclose in a structured data region
- **Unstructured data regions** with the target enter/exit data were added to the standard to handle these cases
- Often, the target enter data will be right after allocation. And the target exit data will be right before the free.

```
float *tmp, *a, *b, *c;
int main(int argc, char *argv[]) {
    int N = 100;
    tmp = (float *)malloc(N*sizeof(float));
    a = (float *)malloc(N*sizeof(float));
    b = (float *)malloc(N*sizeof(float));
    c = (float *)malloc(N*sizeof(float));
    #pragma omp target enter data \
       map(alloc:tmp[:N], a[:N], b[:N], C[:N])
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N);
        saxpy(2.0f, tmp, b, N);

        compute_kernel_2(tmp, b, N);
        saxpy(2.0f, c, tmp, N);

    #pragma omp target exit data map(delete:tmp, a, b, c)
    free(tmp, a, b, c);
}
```

```
void zeros(float* a, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}

void saxpy(float a, float* y, float* x, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

OpenMP® Device Constructs: `target update`

- Issue data transfers to or from existing data device environment
- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],…]
```

- Syntax (Fortran)

```
!$omp target update [clause[[,] clause],…]
```

- Clauses

```
device(scalar-integer-expression)
to(list)
from(list)
if(scalar-expr)
```

Example: target data and target update

```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
  {
#pragma omp target
#pragma omp teams distribute parallel for simd
    for (int i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update to(input[:N])

#pragma omp target
#pragma omp teams parallel for simd reduction(+:res)
    for (int i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i);
  }
```

Review of Data Movement/ Management pragmas and clauses

`map(to:var[0:n]), map(from:var[0:n]), map(tofrom:var[0:n])` can be added to any compute directive

- Structured data regions – a block of code in a functional unit that will have data mapped over at the start and mapped back at the end

```
#pragma omp target data map(to:x[0:n]) map(from:y[0:n])
```

- Unstructured data regions are more flexible and can be done after array creation on the host and just before freeing the memory

```
#pragma omp target enter data map(alloc:var[0:n])
#pragma omp target exit data map(release:var[0:n])
```

- Updating data from host to device or vice-versa

```
#pragma omp target update to(var)   -- host to device
#pragma omp target update from(var) -- device to host
```

GPU and APU Programming model with OpenMP

CPU CODE

```
!allocation on host
ALLOCATE(var(1:N))

!compute on host
!$omp parallel do &
!$omp  private(i), shared(var)
DO i=1,N
   var(i) = …
END DO
!$omp end parallel do
!sync barrier at omp end …
…
!deallocation
DEALLOCATE(var)
```

DISCRETE GPU CODE

```
!allocation on host
ALLOCATE(var(1:N))

!compute on device, expl. mem movement!
!$omp target teams distribute parallel do &
!$omp  map(tofrom:var) private(i),shared(var)
 DO i=1,N
   var(i) = …
 END DO
!$omp end target teams distribute parallel do
!host-device sync barrier at omp end …
…
!deallocation
DEALLOCATE(var)
```

Costly to switch CPU -> GPU!

APU CODE

```
!$omp requires unified_shared_memory
!allocation of unified m
ALLOCATE(var(1:N))

!compute on device, no expl. mem movement!
!$omp target teams distribute parallel do &
!$omp             private(i),shared(var)
 DO i=1,N
   var(i) = …
 END DO
!$omp end target teams distribute parallel do
!host-device sync barrier at omp end …

!deallocation of unified memory
DEALLOCATE(var)
```

Cheap CPU -> GPU copy with APU

- Compute kernel
- Special directive to enable unified memory
- ~~Explicit memory management between CPU & GPU~~ -> not needed for APU!
- Synchronization Barrier

Exercises Introduction to OpenMP Offload

https://github.com/amd/HPCTrainingExamples/tree/main/Pragma_Examples

C/C++ exercises:
Pragma_Examples/README.md (Instructions for Introduction to OpenMP in the first sections)
Pragma_Examples/OpenMP/C/ (Code)

Pragma_Examples/OpenMP/CXX/ (Code)

Pragma_Examples/OpenMP/C/USM (Example code for #pragma omp requires unified_shared_memory and XNACK=0 or 1 behaviour)

Fortran exercises:

Pragma_Examples/OpenMP/Fortran/README.md

Further READMEs in sub-directories with guided exercises with and without XNACK=0 or 1

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


AMD